Use of interaction networks in teaching Minix

Paul Ashton, Carl Cerecke,
Craig McGeachie, Stuart Yeates
Department of Computer Science
University of Canterbury

TR-COSC 08/95, Sep 1995

Use of Interaction Networks in teaching Minix

Paul Ashton* Carl Cerecke Craig McGeachie
Stuart Yeates

Department of Computer Science
University of Canterbury

TR COSC 08/95, September 1995

1 Introduction

Minix is a modular operating system designed for use in teaching of operating systems [3, 4]. In Minix, the operating system is structured as multiple independent processes. These processes communicate with each other, and with user mode processes, via message passing. While the modular structure of Minix simplifies the individual components, the message passing patterns that occur at run-time can be complex.

The interaction network has been proposed as a way of representing all processing (process execution and message passing) that is the direct result of a single user input [2]. An interaction network is an acyclic digraph. Each vertex in the graph represents an event and each edge represents a period of thread execution, or message progression, between two events.

Message patterns are shown very well in displays of interaction networks. In this paper, we use displays of interaction networks to illustrate some of the complex message patterns that occur in Minix. The interaction networks were all recorded for SunOS Minix, a version of Minix that has been ported to run as a SunOS process [1].

The interaction networks selected show the message passing that occurs in the following situations:

- Input of character that is part of a command line.
- Process creation and termination.
- Execution of a new process image.
- Two user processes communicating through a pipe.

We show the value of the interaction network as a tool for understanding distributed processing by using it to provide insights into the messages exchanged in a number of situations that frequently arise in Minix.

*e-mail: paul@cosc.canterbury.ac.nz

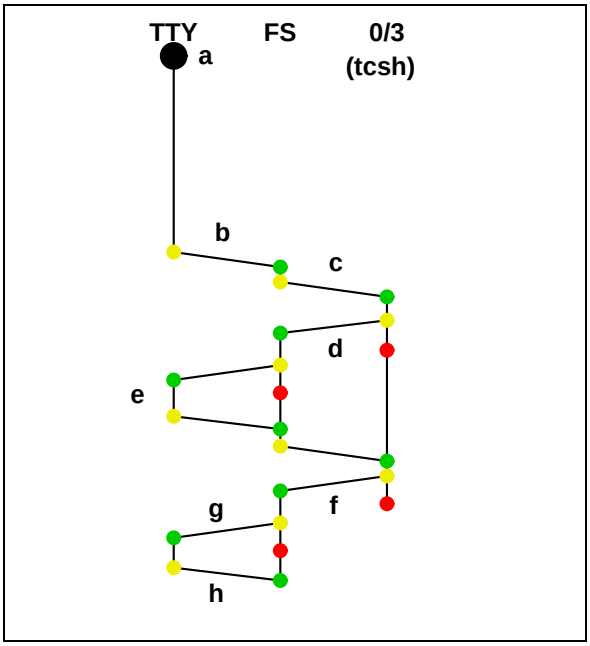

Figure 1: Interaction network showing reaction to input of a single character that is part of a shell command line

2 Case study 1: input of a single character

The interaction network in Figure 1 shows the processing that occurs as the result of input of a single character that is part of a shell command line. The shell in use is the `tcsh`, a shell that reads command lines character-by-character to allow for command line editing. The character whose input triggered the interaction network did not terminate the command line, so all that occurred was that the `tcsh` echoed the character and added it to a buffer.

Before discussing this example further, some general information on interaction network displays may be useful. The vertices (each representing an event) are arranged in columns, with one column for each process that performed processing in response to the user input. Time increases as you go down the network, with the Y-coordinate of each vertex in direct proportion to the time at which the event represented by the vertex occurred. The elapsed time between the initial (top-most) and final (bottom-most) vertices in Figure 1 is 0.123 seconds. All of the vertical edges represent activities carried out by processes, and (in this figure anyway) each non-vertical edge represents a message. The length of each edge in the Y direction shows the duration of the activity that it represents.

2.1 Case study 1: step-by-step explanation

1. Vertex `a` represents the moment at which the terminal task was notified that a character had been typed. This causes the terminal task to copy the character into a buffer in the address space of the `tcsh` process.

2. The terminal task sends a REVIVE message (`b`) to the file system to tell it that a

process that was suspended waiting for terminal input can resume execution.

3. The file server sends message `c`, which is a reply to an earlier READ request by the `tcsh` shell. On receiving message `c`, the shell transfers the character to the command line it is building up.

4. The shell sends a WRITE message (`d`) to the file system so as to echo the character just entered.

5. To perform the echo, the file system sends a TTY_WRITE message to the terminal task, which then replies that the character has been echoed successfully (`e`). The file system then relays this fact to the shell in a reply message.

6. As the shell has not yet got a complete command line, it sends a READ message (`f`) to the file system asking it to read further terminal input.

7. The file system sends a TTY_READ message to the terminal task asking for terminal input.

8. There is no terminal input, so the terminal task replies to that effect with message `h`.

2.2 Case study 1: discussion

1. Minix uses a rendezvous message passing protocol. In rendezvous, a message is copied from sender to receiver only when both are ready to communicate. The process that becomes ready first must block, and await the other. Several of these block events can be observed in Figure 1. In each case, a process sends a request message and then blocks waiting for a reply. For example, after the shell sends message `d` it immediately tries to receive a message from the file system, and blocks because the file system is not trying to send a message to the shell. In this case the receiver is blocking.

   One consequence of the use of rendezvous is that the message passing activities shown in Figure 1 have very short durations (typically a little over 3 milliseconds). This is because all that is involved is copying the message from the address space of one Minix process to the address space of another. If a message passing protocol were used that allowed messages to be buffered within the kernel, then some messages would exist for much longer periods.

2. The processing towards the bottom of the interaction network is interesting. When the terminal task discovers it has no characters to return to the file system, it replies anyway. This allows the file system to continue processing requests. If the terminal task did not reply until it had input, then the file system (because it is single threaded) would be suspended for a potentially very long period. If the file system is suspended for a considerable period, then very quickly all active processes would find themselves waiting to rendezvous with the file server.

   Notice that the file server does not reply to the shell. This keeps the shell suspended until there is further input. In Minix, the only process queues are the ready-list and those associated with message passing.

   When subsequent input does occur, the terminal task sends a REVIVE message to the file system that allows it (finally) to reply to the earlier READ. So message `c` is a reply

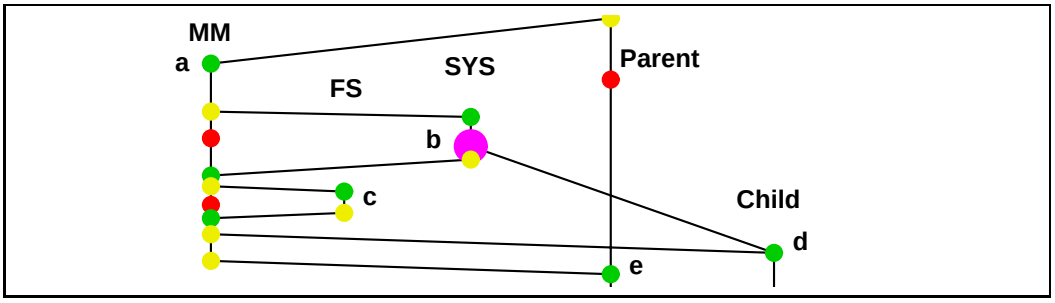

Figure 2: Interaction network fragment showing message that results from a `fork` system call

to the message `f` that was sent in the previous interaction network recorded for this user. It should be clear how multiple instances of this interaction network would occur one after the other as the characters of a command line were entered, with the message passing towards the end of one interaction network setting the scene for the message passing that occurs at the start of the next.

3. Nine messages are passed just to put one character into a buffer and echo it. In fact a tenth message is involved at the start of the interaction. It is sent by an interrupt handler to the terminal task to inform the terminal task that input has arrived. This message does not appear in Figure 1 because in the current version of the monitor the probe that detects terminal input is in the terminal task.

3 Case study 2: creation of a process

Process creation requires cooperation between all of the major operating system components. The interaction network fragment in Figure 2 shows the message passing that results from a `fork` system call.

3.1 Case study 2: step-by-step explanation

1. In this part of the interaction network, the first message shown is a FORK message sent by the parent process to the memory manager (`a`) asking that it create a new process. The memory manager allocates memory for the new process (amongst other things).

2. The memory manager informs the system task of the new process by sending a SYS_FORK message (`b`).

3. The memory manager informs the file system of the new process by sending a FORK message to it (`c`).

4. Process creation is now complete. The memory manager sends a reply message to the child (`d`) with a return value of 0, and a reply message to the parent (`e`) with a return value of the process id of the child process.

3.2 Case study 2: discussion

1. In Minix, the process table is split vertically in three. One part is maintained by the kernel (which consists of all interrupt handlers and device driver processes), a second by the memory manager and a third by the file system. The memory manager coordinates the creation of a new process, and creates a new entry in its own process table. It must send messages to the system task and the file system so that appropriate entries are created in the process tables of the kernel and the file system.

2. The child process is shown as being created by the system task. The edge from the creation event to the reception of message `d` represents a period during which the process was blocked waiting to receive a reply from the memory manager (which is precisely what the parent process was doing at the time the child was created).

3. The message passing in this case is rather unusual. A single request to the memory results in two replies—one to the parent, and one to the child.

4. The `fork` system call has given rise to 7 messages.

4 Case study 3: loading a new program image

The `exec` system call, which causes a process to execute a newly loaded program, gives rise to a considerable amount of message passing. Much of the interaction network fragment shown in Figure 3 is caused by one `exec` system call, and this will be discussed first. Following this will be discussion of two related system calls—`wait` and `exit`.

4.1 Case study 3: `exec` step-by-step explanation

1. The first message in Figure 3 sent by the user process running the program `exec.c` is an EXEC message (`a`) sent to the memory manager requesting that a new program image be loaded for that user process.

2. The memory manager needs the pathname of the executable program, so it asks the system task (`b`) to copy the pathname from the address space of `exec.c` to the address space of the memory manager.

3. The memory manager sends a CHDIR message to the file system to get it to change the memory manager's current working directory to that of the `exec.c` process.

4. The executable file is opened (`d`) by sending an OPEN message to the file system.

5. An FSTAT is message is sent to the file system to get information on the newly-opened file (`e`). The file type and protection mode are needed to make security checks, check that the file is executable, and to see whether the executable file is set-uid and/or set-gid.

6. The memory manager changes back to its own current working directory (`f`).

7. The header (containing the sizes of the various sections) of the executable file is read into memory (`g`). The DISK task is sent a SCATTERED_IO message to have it transfer

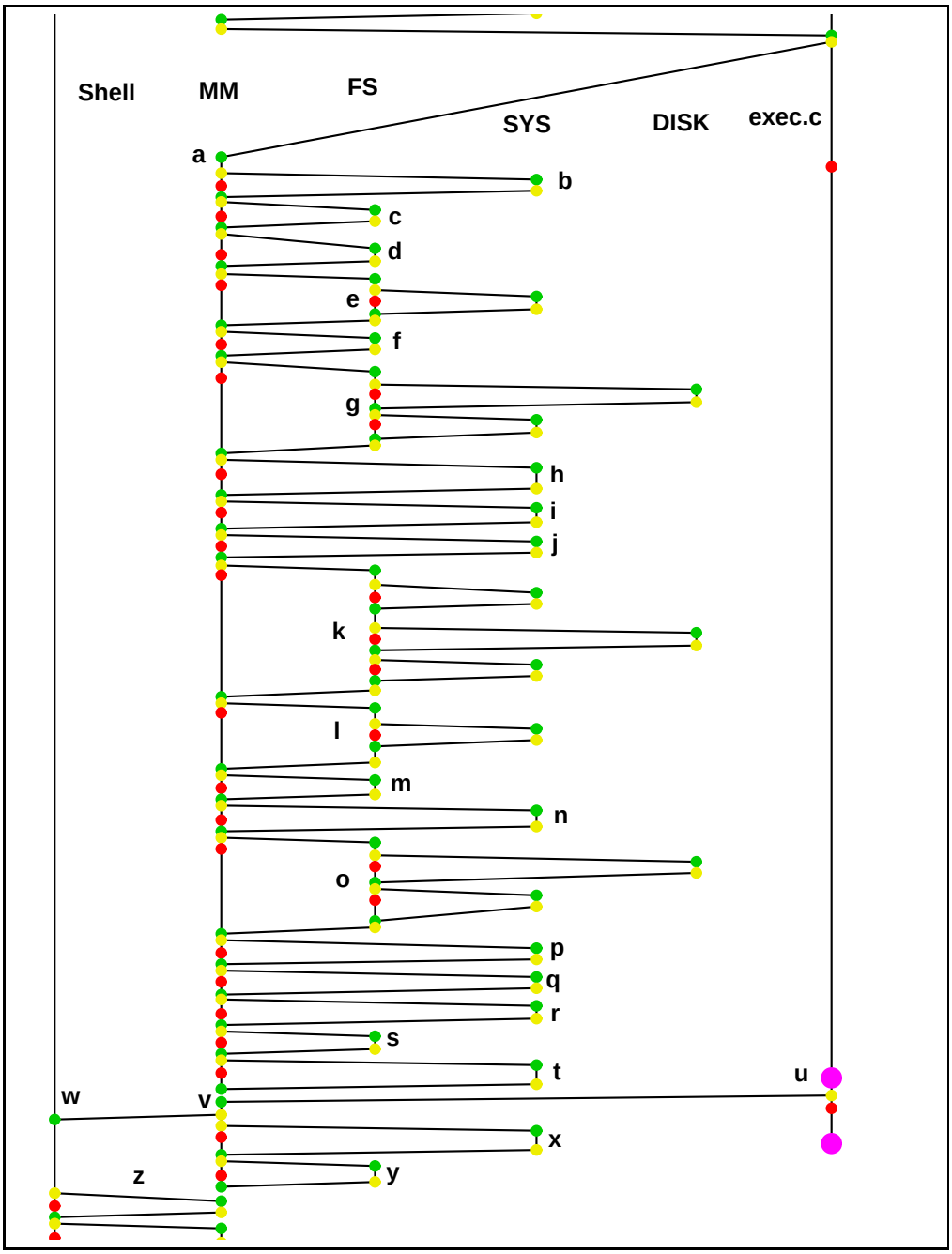

Figure 3: Interaction network fragment showing message passing that results from `exec` and `exit` system calls

a 1024 byte block into the file system's cache area, and the system task copies the header portion of that block to the memory manager.

8. The stack of the process that invoked `exec` is copied into the memory manager (`h`). The copied stack contains the command line arguments and environment that are to be supplied to the program being loaded.

9. The memory manager uses information in the header to compute the amount of memory needed by the new executable. The appropriate amount of memory is allocated, and the old memory area is released. The kernel is informed of the address space changes via a SYS_FRESH message to the system task (`i`).

10. A newly constructed stack, containing the command line arguments and the environment, is copied into the new address space (`j`).

11. A single READ request is sent to the file server so that the text segment is transferred into the new address space. The file system reads in the text segment in 1Kb blocks (`k`). The first block is already cached by the file server because it contains the header. The second block is still on disk.

12. The data segment is transferred into the new address space (`l`) in 1Kb blocks.

13. The memory manager sends an LSEEK message to step over the symbol table in the executable (`m`).

14. Text and data relocation must now occur, because in SunOS Minix all processes run in the data segment of a SunOS process, and the starting address of a loaded program is not determined until run-time. First the text segment is relocated. A SYS_MPROT message (`n`) is sent to the system task to make the text segment accessible to the memory manager. The SYS_MPROT message was added to SunOS Minix as part of a memory protection subsystem.

15. The text relocation information is read into the memory manager from the executable file (`o`).

16. The relocations are performed, then another SYS_MPROT message is sent (`p`) to disable access to the text segment.

17. Data relocations now occur. In fact there are none, but the messages to enable and disable access to the data segment (`q` and `r`) are still sent (room for some optimisation here).

18. The memory manager has finished with the executable file, so it is closed (`s`).

19. The new executable has been loaded. The memory manager sends a SYS_EXEC message (`t`) to the system task. This causes the system task to assign values to some of the saved registers of the process that has just loaded the new program. The main registers set are the program counter and the stack pointer. The system task also marks the process as runnable, and records a new "name" for the process for use in kernel debugging reports. The new name event is shown as an event in `exec.c` (`u`).

4.2 Case study 3: `exec` discussion

1. The memory manager and the file system cannot directly access the address spaces of any other process. They need to send SYS_COPY messages to the system task to ask it to do the copying on their behalf. This is apparent from the memory manager having to get the system task to copy the pathname of the invoked program and to copy various program stacks, and the file system having to get the system task to copy file data to and from other address spaces.

   Use of SYS_COPY in performing relocations would have been very clumsy, so SYS_MPROT messages are used to make the segment being relocated writable for the period during which relocation is performed.

2. The memory manager (temporarily) changes its current working directory to that of the process that sent the EXEC message so that if the pathname of the file to execute is relative to the current working directory, then the pathname is interpreted relative to the correct directory.

3. During the processing of the CHDIR message by the file system, you might have expected to see use of the system task to copy a directory name. The file system handles CHDIR requests from the memory manager in a special way—it simply copies a pointer to a structure that contains information about a directory. In the first call (`c`) the pointer is the current directory pointer of the specified process, and in the second (`f`) it is the file system's root directory pointer.

4. During the processing of the OPEN by the memory manager (`d`), you might have expected to see use of the system task to copy the name of the file to open. The system task is not needed in this case because the file name is short, and for OPEN messages short file names are stored directly in the OPEN message.

5. The three disk accesses have short elapsed times (of the order of 4 milliseconds). In SunOS Minix, disk reads are simulated by issuing SunOS read system calls. In all of these cases it is highly likely that SunOS had the information already cached, so no access to a physical disk was required.

6. When the memory manager has finished performing the `exec` call, it does not reply (if you look closely, you can see that the last event in the memory manager that is part of the `exec` handling is the receipt of message `t`). This is because the user process that originally invoked `exec` re-starts execution at the beginning of the new program, and is not expecting a reply to an EXEC message sent by an earlier program.

7. A total of 57 messages were sent during processing of the EXEC system call. This number is not a constant—it depends on the size of the executable file, and the extent to which the executable is cached by the file system. Also, the fact that relocation is required is a consequence of use of a single address space (as is also the case in the versions of Minix for 68000 machines), and the SYS_MPROT messages are a peculiarity of SunOS Minix.

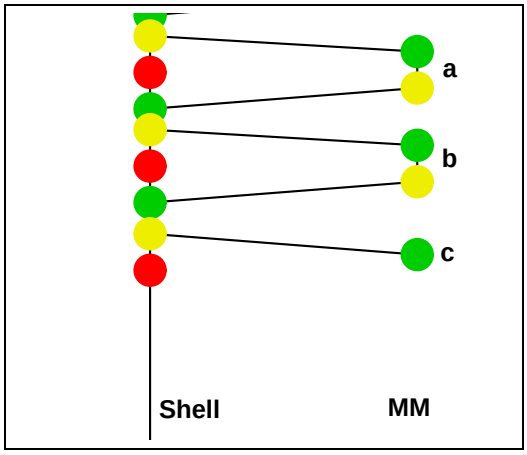

Figure 4: Interaction network fragment showing the start of a `wait`

4.3 Case study 3: `exit` step-by-step explanation

Also illustrated in Figure 3 is the message passing that results from the `exit` system call. The program whose loading is shown in the upper part of the figure simply exits immediately.

1. To terminate itself, the `exec.c` process sends an EXIT message to the memory manager (`v`).

2. The shell has been waiting for one of its child processes (such as the one that it created to run `exec.c`) to terminate. As one of the shell's child processes has just exited, a reply message (`w`) to a previous WAIT is sent.

3. The system task and file system are informed of the termination of the process so that they can update their process tables accordingly (`x` and `y`). Note the symmetry with the message passing associated with the creation of a process.

4.4 Case study 3: `exit` discussion

1. Again (as with EXEC), no reply message is sent to the EXIT message (a gap can be seen between the memory manager receiving reply `y` from the system task and receiving message `z` from the shell). In this case there is no process to reply to.

2. The beginning of the shell's wait occurs earlier in the interaction network, and is shown in Figure 4. The memory manager receives the WAIT message (`c`), but finds that the shell must be suspended until one of its child processes exits (messages `a` and `b` are SIGNAL messages, and are the last system calls made by the shell before it begins waiting for a child to exit). Once again we see a process being suspended because a reply message is withheld.

3. Once awakened, the shell tidies up so that it can prompt for the next command. The first thing it does is to send a SIGNAL message to the memory manager (`z`).

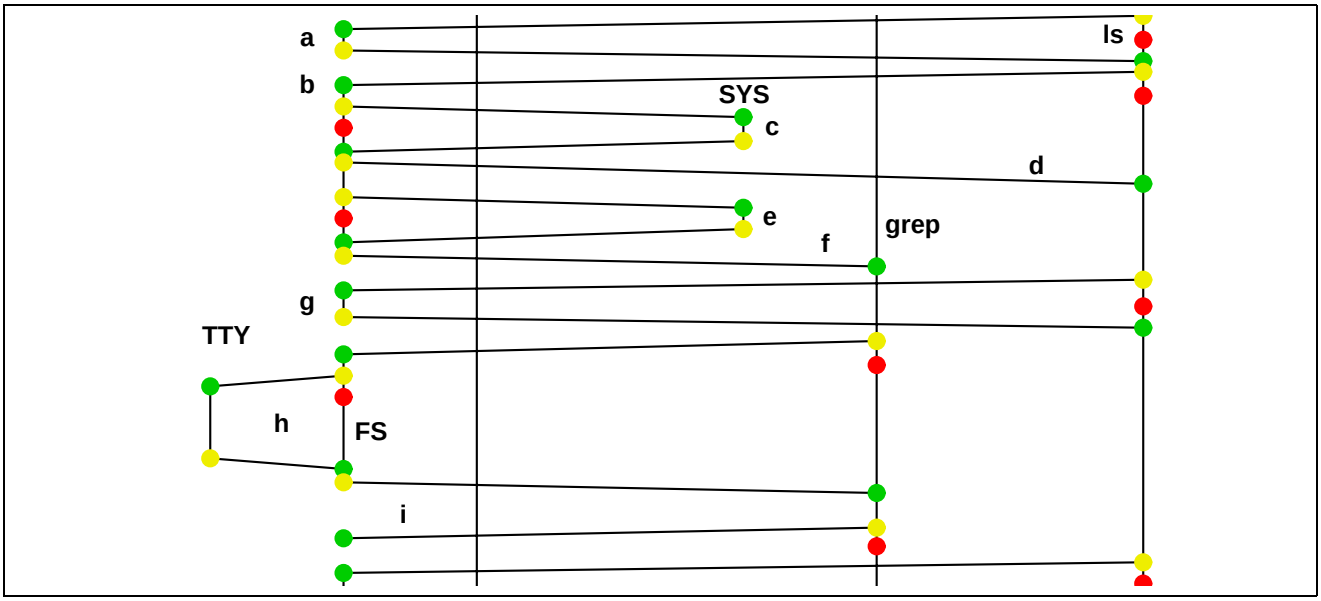

Figure 5: Interaction network fragment showing communication via a pipe

5 Case study 4: communication through a pipe

The only way for two Minix user processes to exchange messages is through a pipe. Figure 5 contains an interaction network fragment that shows one buffer progressing through a pipe. The interaction network resulted from the entry of a newline at the end of the command line `ls -la /usr/bin | grep -`

5.1 Case study 4: step-by-step explanation

1. At the top of the figure we can see the `ls` process sending its last message (`a`, a CLOSE message) before it writes to the pipe.

2. `ls` writes to the pipe by sending a WRITE message to the file system (`b`).

3. The file system gets the system task to copy data from the `ls` process into a buffer set aside for the pipe (`c`).

4. The write is complete, so the filesystem replies to `ls` (`d`).

5. Having replied to the WRITE, the file system checks to see whether there are any processes suspended, reading from the pipe. It finds there is one, so the file system gets the system task to copy the data from the pipe to a buffer in the `grep` process (`e`).

6. The file system then sends to `grep` a reply to an earlier READ message (`f`).

7. `ls` starts producing its next output by sending an OPEN message to the file system (`g`).

8. On receiving lines of input, `grep` determines that (at least some of) the lines match the pattern it is searching for, and it writes the matching lines (`h`). Writing the lines

involves sending a WRITE message to the file system. The file system determines that the specified file descriptor is associated with a terminal, so sends a TTY_WRITE message to the terminal task.

9. `grep` sends a READ message to the file system (`i`) to request further input from the pipe. The pipe is empty, so the file system goes on to receive the next message without replying to the READ, thereby leaving `grep` suspended.

5.2 Case study 4: discussion

1. Again note that, as with the message passing in Figure 1, the message passing pattern shown in Figure 5 will be repeated several times as data flows through the pipe. Later in the interaction network that is partly shown in Figure 5 `ls` writes to the pipe, causing an `f`-like message to be sent to `grep` as a reply to message `i`.

2. Sending one message through the pipe has involved eight Minix messages.

3. Once again a process is blocked by simply not sending a reply to it.

6 Conclusions

The interaction networks and network fragments that appear as figures in this document have shown several complex message passing patterns that arise in Minix. The figures give considerable insight into the internal operation of Minix, and show that displays of interaction networks are useful teaching tools.

7 Acknowledgements

This paper resulted from work done as part of an assignment set by the first author for a fourth year honours class. The other authors were the members of that class whose assignments contained much of the raw material for this paper. Other members in the class (Mark Alexander, Andrew Bryant, Mark Dunlop, Simon Knudsen, Justin Macfarlane and Chris Tsui) also did good work in the assignment, many producing case studies similar to those included here.

References

[1] Paul Ashton, Daniel Ayers, and Peter Smith. SunOS Minix: a tool for use in operating system laboratories. In *ACSC-17: Proceedings of the Seventeenth Annual Computer Science Conference*, pages 259–269, Christchurh, New Zealand, January 1994.

[2] Paul Ashton and John Penny. A tool for visualising the execution of interactions on a loosely-coupled distributed system. *Software—Practice and Experience*, (accepted for publication).

[3] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[4] Andrew S. Tanenbaum. A UNIX clone with source code for operating systems courses. *Operating Systems Review*, 21(1):20–29, January 1987.